From: Moaaz Zaki
Date: 13 March, 2024

Node classification with Graph XGBoost

This report addresses the problem of predictive node classification in directed graphs, where each graph consists of nodes and edges representing directional relationships between them. Some nodes within these graphs are missing, and the objective is to develop a predictive algorithm capable of determining the types of the missing nodes based on the existing graph structure.

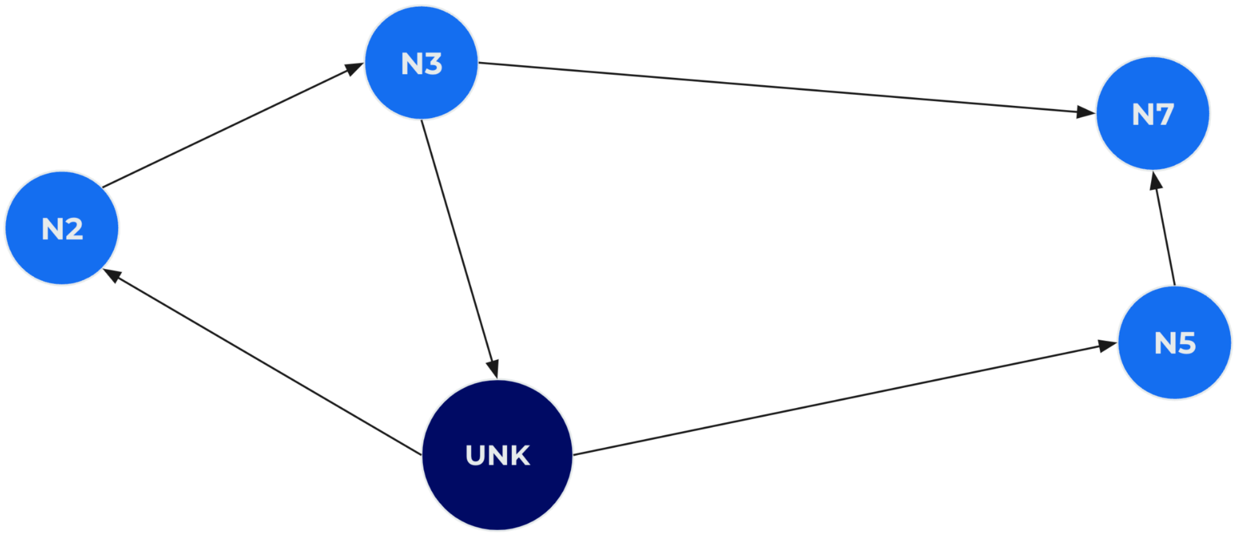

Figure 1. Example Graph with missing nodes

Dataset

By examining the provided dataset, it comprises around 10,000 graphs. The dataset is divided into three splits: 60% for training, 20% for validation, and another 20% for testing, as illustrated in *Figure 2*.

The dataset encompasses 40 classes ranging from N0 to N39. The distribution of each class among the three subsets of the dataset (train, validation, and test) is nearly identical.

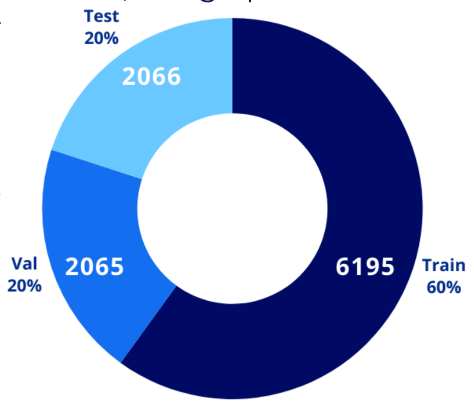

Figure 2.Subsets of the dataset

By examining the distribution of node types, the percentages of splits for each node type are approximately equal. Similar observations are evident for the number of nodes per graph and number of edges per graph. For a visual representation of these distributions across the train, validation, and test subsets, refer to *Figure 3*.

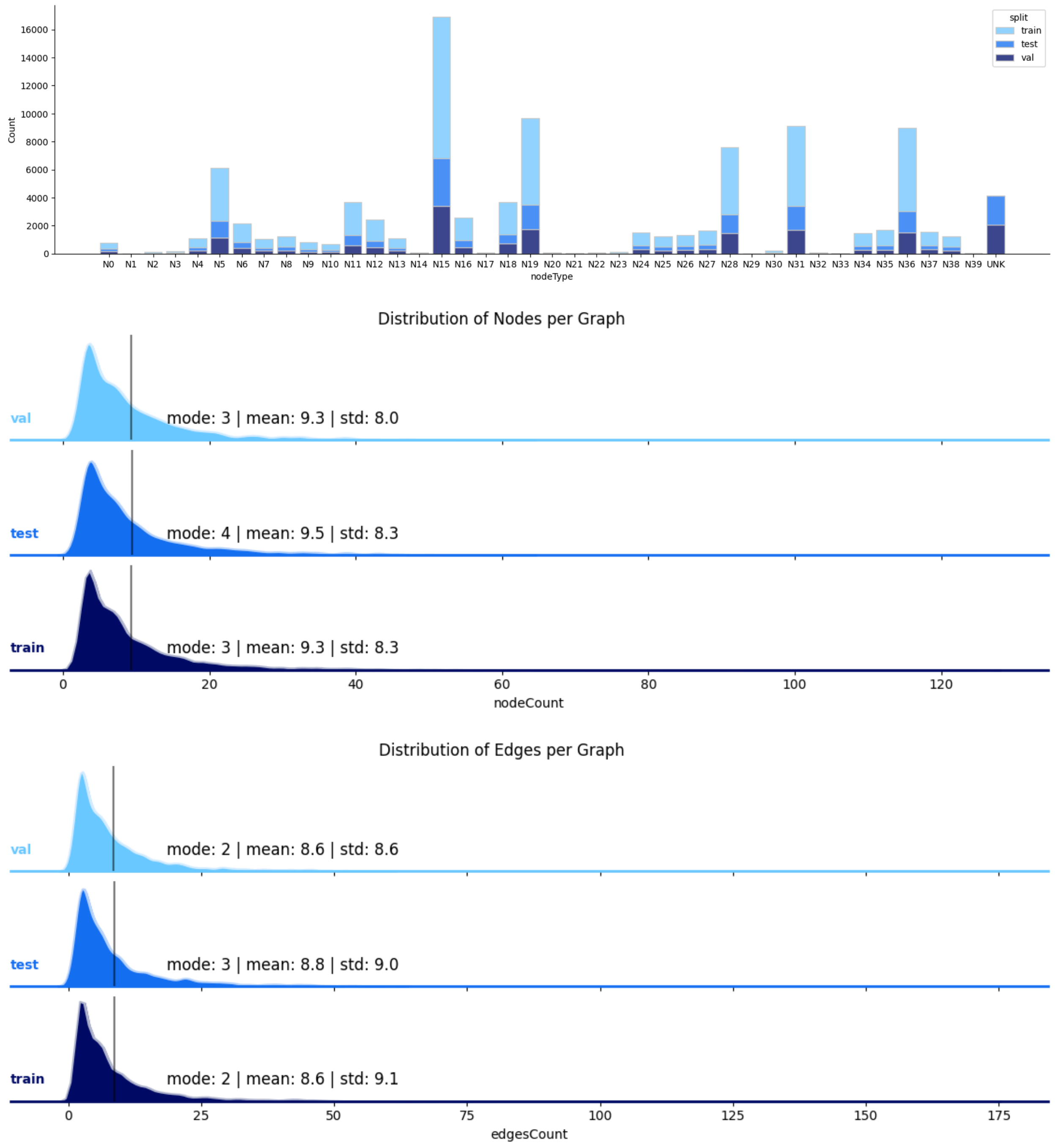

Figure 3. Nodes types, nodes per graph, and edges per graph distributions across train, validation and test subsets

The dataset exhibits a noticeable class imbalance, prompting the inclusion of tree models in our experiments. Tree models are chosen for their inherent ability to handle bias in class imbalances. However, it's worth noting that approximately 12 node types (e.g., N1) are nearly absent in the dataset. This scarcity may lead to suboptimal performance for the corresponding node types in our analyses.

Proposed solution

As discussed in the previous section, tree-based models emerge as strong candidates for effective predictive algorithms. However, when devising an approach for developing a robust predictive algorithm, certain key considerations come to the forefront:

1. **Node Features**: Given that the nodes in the graph are non-attributed, effective feature extraction must encapsulate information from the graph, edges, and nodes to represent each node as a vector. It is crucial that this vector avoids incorporating implicit information about the current node type, as such information is typically unavailable for unknown nodes in practical scenarios.

2. **Learning Strategy**: Knowing the fact that certain node types are almost absent in the dataset, an adaptive learning strategy becomes essential. Incorporating the concept of incremental learning is advisable. This involves connecting the generated model to a human/AI feedback loop that functions as a teacher. The model can then learn from new feedback, continually improving its understanding of both seen and unseen patterns over time. This adaptive learning approach ensures the model's responsiveness to evolving data dynamics.

The proposed solution endeavors to address all the crucial points mentioned earlier while maintaining simplicity and incorporating automated improvement through incremental learning. The following figure (Figure 4) illustrates the pipeline and the technologies employed in creating a reliable, lightweight, and accurate approach that evolves dynamically over time. This approach aims to strike a balance between comprehensiveness and ease of maintenance, ensuring its adaptability and effectiveness as it continues to learn and improve.

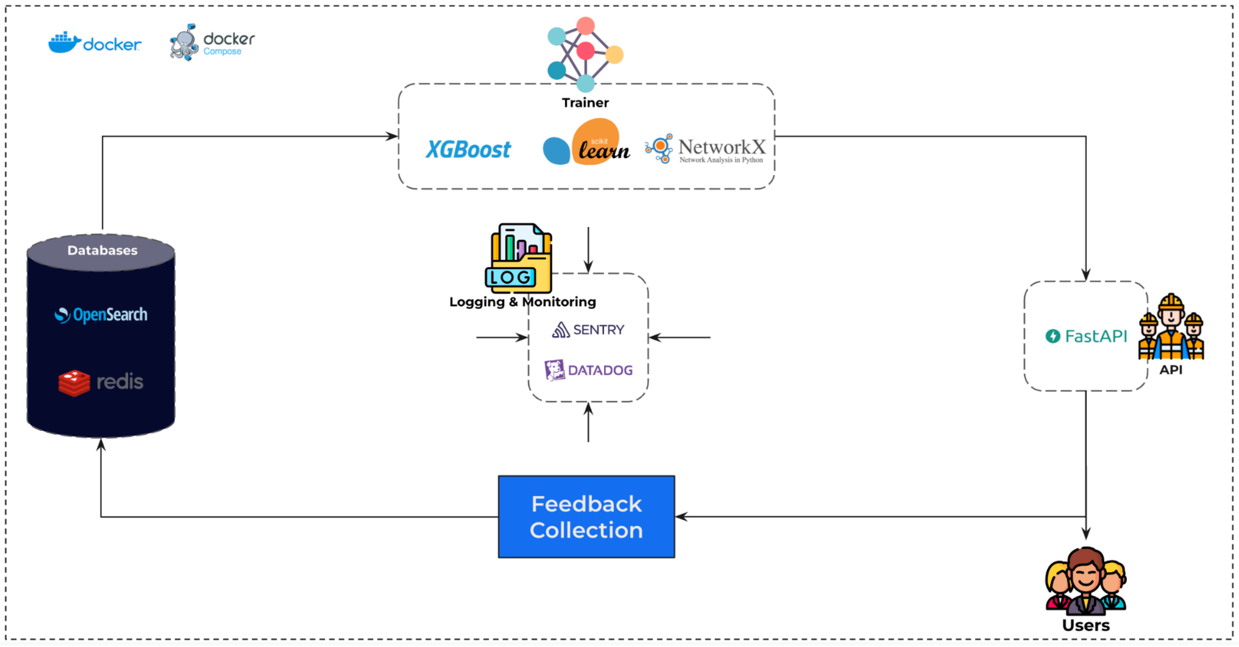

Figure 4. Pipeline and technologies used per component for the proposed approach

The illustrated solution in Figure 4 comprises the following key components:

1. **Databases**: This component is dedicated to temporarily caching recently predicted results of the model using Redis. Subsequently, the answers are stored permanently upon being rated (corrected) by the teacher (User/AI) through opensearch.

2. **Trainer**: The trainer automatically gathers stored answers once a sufficient new set is accumulated (typically 1k new graphs). It then generates a new model by fitting an XGBoost model with L1 regularization, subsampling, and tree pruning. The model is designed to adapt to class imbalance and mitigate overfitting as much as possible.

3. **API**: This is the application interface enabling users to submit graphs with missing nodes. Users receive predictions for the missing nodes and can optionally provide feedback to enhance the model.

4. **Logging & Monitoring**: This component facilitates debugging by enabling the identification of system issues on a per-component basis. It also provides a comprehensive view of system performance and health.

Let's reconsider the challenge of representing non-attribute nodes in a graph. The proposed solution suggests generating a vector for each node, incorporating both positional and structural characteristics within the graph. The features derived from this approach include:

1. **Degrees**: Total, in, and out degrees for each node, conveying its connectivity within the graph.

2. **Betweenness Centrality**: measures the extent to which a node lies on paths between other vertices. Vertices with high betweenness may have considerable influence within a network by virtue of their control over information passing between others.

3. **Eigenvector Centrality**: Quantifies a node's influence by assigning relative scores based on connections to other high-scoring nodes in the network.

4. **Page Rank Centrality**: Similar to Eigenvector Centrality but considers link direction, weighting incoming links based on the originating node's score.

5. **Katz Centrality**: Another variant of Eigenvector Centrality, considering the total number of walks between node pairs to gauge influence.

6. **Closeness Centrality**: Reciprocal of the sum of shortest path lengths between a node and all others, indicating how central a node is within the graph.

7. **Graph Density**: Ratio of present edges to the maximum possible edges, offering insights into the graph's connectivity density.

8. **Local Clustering Coefficient**: Measures how close a node's neighbors are to forming a clique (complete graph), reflecting the tendency of nodes to cluster in the graph.

9. **Types of Neighbor Nodes**: A vector with dimensions equal to the number of classes, counting how many times a specific type serves as a neighbor for a given node.

10. **Types shortest paths**: A vector representing the count of 16 possible directed subgraphs of 3 nodes, providing information on graph density and transitivity.

11. **Graph Motifs**: A vector of the number of 16 possible directed subgraphs that consist of 3 nodes, this feature gives information about graph density and transitivity.

By combining these features, a vector representation is obtained, depicting how a node is situated in the graph with known properties, along with some insights about its interactions with other node types.

Results

The proposed approach is tested by first training on only the training set, and testing on validation, the results are the following for this setup:

Accuracy	WA* Precision	WA* Recall	WA* F1
65%	66%	64%	64%

Table 1. Results of only including training set

* WA: Stand for weighted average, which is averaging the metric over individual scores of each class while taking the class frequency into consideration

For further analysis, F1 score percentage is shown in the following figure (Figure 5) for each node type, including the types that were missing from the validation set.

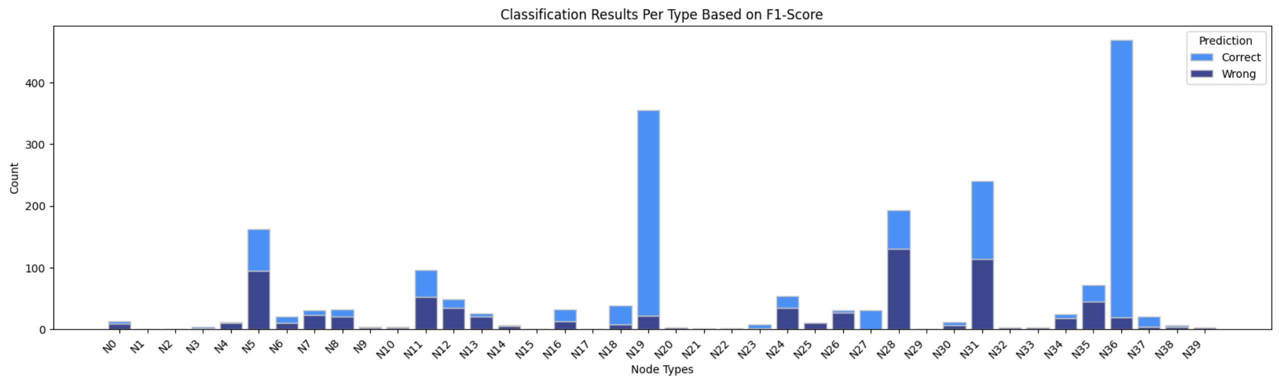

Figure 5. F1 percentage across different types

After that, to demonstrate the ability of incremental learning of the model, a subset of validation data is picked to incrementally enhance the knowledge of the model, while testing against the unseen subset, the results are the following:

Accuracy	WA* Precision	WA* Recall	WA* F1
69%	71%	66%	68%

Table 2. Results of including training set and a portion of the validation set

* WA: Stand for weighted average, which is averaging the metric over individual scores of each class with taking the class frequency into consideration

Deployment Process

The pipeline is packaged as a unified **cluster of discrete containers**, each designated for specific roles such as the Python FastAPI service, trainer, Redis, OpenSearch, and the Datadog agent. These containers collectively form a cohesive unit. Deployment is facilitated through **Docker Compose**, enabling smooth integration with **AWS ECS**. Alternatively, the configuration can be adapted for Kubernetes Resources using the user-friendly "Kompose" tool.

An important feature of this setup is its **scalability.** The cluster is designed to easily scale the number of running containers per service, providing flexibility to adapt to varying workloads through **Ngnix traffic handling and load-balancing**. While the deployment is configured for convenience, it's advisable to deploy OpenSearch as a separate cluster, complete with persistence storage for potential utilization in other projects. It's crucial to recognize that this arrangement primarily serves as a demonstrative configuration, showcasing the operational stack of the system with the added capability to scale up container instances as needed.